Architectural models of J2EE Web tier frameworks

Timo Westkämper

University of Tampere
Department of Computer Science
Master's Thesis
May 2004

University of Tampere
Department of Computer Science
Timo Westkämper: Architectural models of J2EE Web tier frameworks
Master's Thesis: 57 Pages
May 2004

Abstract

The purpose of this thesis is to describe the architectural models of four selected Java 2 Enterprise Edition (J2EE) Web tier frameworks. The approach to the subject is a conceptual-analytical research with a focus on the application concerns the different frameworks promote. The thesis begins with an introduction to frameworks and the basic categorization into *Black-box* and *White-box* frameworks. An introduction to the J2EE with a focus on the Servlet framework and the general J2EE application model follows. After that the chosen frameworks are presented. *JavaServer Faces* and the *Portlet* framework as frameworks from Sun Microsystems and *Jakarta Struts* and *Apache Cocoon* from the Apache Software Foundation cover most of the current framework extensions for the J2EE Web tier. The thesis concludes with a discussion of the results and presents a compact evaluation of the frameworks with a focus on possible future scenarios.

Keywords: J2EE, Web tier, Presentation tier, Frameworks, Jakarta Struts, JavaServer Faces, Apache Cocoon, Java Portlet

Table of Contents

Acknowledgements

My respect and honest appreciation belong to my two supervisors Mr. Roope Raisamo and Mr. Isto Aho from the University of Tampere, who guided me in the formal aspects of this thesis and the shaping of the final subject, and Mr. Zhiguo Guo, who was a great help in discussing and reviewing the technical aspects of this thesis. Thank you all!

1. Introduction

With the advance of the Internet and especially the World Wide Web in the early 1990s and its increasing importance in *Business-to-Customer* and *Business-to-Business* services, the need to define standard methodologies to design and build efficient, robust and maintainable Web applications has become increasingly important. Amongst others the Enterprise Edition of the Java platform (J2EE) offers a technical platform and set of frameworks to design web applications and services. In the standard three-tier application model of the J2EE the web application issues are located in the *Presentation tier* or more specifically in the *Web tier*.

As the J2EE is a relatively young technology platform, there has not yet been much academic reflection on it. Most of J2EE-related research concentrates on the *Business tier*, such as for example "Distributed Component Technologies and their Software Engineering Implications" [Emmerich, 2002]. Some interesting examples of non-academic research on J2EE Web tier frameworks are the Wafer project [Wafer, 2004], a comparison of J2EE Application frameworks, and the paper related to the JavaOne presentation "Navigating the Application Development Frameworks Terrain" [Cann, 2003]. Concerning the J2EE aspects of my paper, the J2EE specifications and various articles in online newspapers have been more useful than results of academic research.

General research on frameworks has also proven to be useful in the context of the chosen subject. Good overviews of framework types and purposes helped to see the J2EE frameworks in a wider context. I will refer to the papers "Object-Oriented Frameworks [Mattsson, 1996], "Design, implementation and evolution of object oriented frameworks" [vanGurp, 2001] and "Object-Oriented Application Frameworks" [Fayad, 1997] in later chapters as they provide a good theoretical basis to categorize and analyse frameworks.

The primary problem domain of this thesis is the *J2EE Web tier frameworks*, which came up quickly after the introduction of the *Servlet framework*. The Servlet framework provides a basic component framework for web application development, whereas the Web tier frameworks address several issues of web application development that were not covered by the first and subsequent versions of the Servlet framework. Such issues are for example models of *Navigation* and *Page flows*, *Personalization of Services*, *Publication* of heterogeneous content and more specific *Security declarations*. Another important function of the Web tier frameworks is to enforce certain application architectures, which are only suggested in the initial specifications of the J2EE Architecture, such as the Web tier design patterns of the J2EE BluePrints.

The personal motivation for writing this thesis is to gain detailed insight into some parts of the Java 2 Enterprise Edition and from a more academical perspective it is the fact that there has not been much research in this area. Research on the Java 2 Enterprise Edition concentrates mostly on Enterprise Java Beans, which is much less open for direct external influences compared to the J2EE Web Tier technologies.

Java as a platform is entirely software based. It is based on the Java Virtual Machine and runs on top of hardware-based platforms. Each hardware platform needs to have its own Java platform implementation to run Java applications. The three editions of the Java platform are the *Java 2 Standard Edition* (J2SE) for general applications and desktop GUI programming, the *Java 2 Micro Edition* (J2ME) for mobile computing and the *Java 2 Enterprise Edition* for enterprise and distributed systems. In the following I will present each of them with a special focus on the Java 2 Enterprise Edition [O'Reilly, 2002].

The *Java 2 Standard Edition* (J2SE) provides the main elements for Java development. It is the basis for the Enterprise Edition (J2EE) and Java Web Services technologies, but provides mainly frameworks for the development of Java desktop and stand-alone applications. Applets, Beans, Networking APIs, Threading, Media APIs, Java Foundation Classes (Swing/AWT) and Security are the main elements of the Standard Edition.

The *Java 2 Micro Edition* (J2ME) is a set APIs and Frameworks targeted at mobile and embedded devices. Mobile phones, PDAs (Personal Digital Assistants) and printers are examples of devices where the Java 2 Micro Edition can be deployed. The Personal Java API, the Java TV API, the Embedded Java API and the Java Card API are the main elements of the Java 2 Micro Edition.

The *Java 2 Enterprise Edition* (J2EE) is targeted for use in enterprise application development. Interoperability through Web Services (SOAP, XML-RPC), component technologies (CORBA, COM, EJB) and connectivity to database and legacy systems (JDBC, Connection API) are the main concerns of the Enterprise Edition. The Servlet Framework as Java's CGI abstraction (Common Gateway Interface) and various XML APIs conclude the palette of frameworks. The frameworks can be roughly divided into two categories. Web Services, the XML APIs, the Connection and JDBC APIs and Component Interfacing as *Communication Services* and Servlets as well as Enterprise Java Beans as *Component frameworks*.

The body of this thesis is divided into nine chapters. The first introduces the subject and gives a broad overview of the contents of this thesis. The second chapter gives a basic introduction to frameworks and introduces the

categorization into Black-box and White-box frameworks. In the third chapter the technologies of the J2EE Web tier are introduced. The fourth chapter discusses briefly some methodological issues of this thesis. The fifth to eighth chapter are used to give overviews of the J2EE Web tier frameworks. While the primary concerns of the overviews are architectural issues, deployment concerns and evolutionary aspects are discussed as well as secondary issues. The two final chapters conclude this thesis with a discussion of the results and a brief summary.

2. Introduction to frameworks

This chapter offers a basic introduction to object-oriented frameworks, their benefits and one possible general categorization of frameworks. Frameworks are reusable semi-complete application constructs that can be used to build complete applications. They can be seen as the object-oriented equivalent to libraries.

2.1. Benefits of using frameworks

Frameworks in general or more specifically object-oriented application frameworks are often built on top of basic design pattern implementations to enforce and facilitate certain development styles. The main benefits of object-oriented application frameworks are *Modularity*, *Reusability*, *Extensibility* and *Inversion of Control* [Fayad, 1997]. In the following these aspects will be described in detail.

Frameworks enhance *Modularity* by providing stable interfaces and hiding implementation details. The applications which are built on top of it have to conform to the interfaces provided by the framework. True modularity is rarely achieved as frameworks often include various controlling mechanisms in their classes and interfaces, which makes it difficult to replace entire areas of the framework by application-specific versions.

Concrete *Reusability* of individual elements is also a design goal rarely achieved in pure fashion. In many frameworks the construct can be (re)used as a whole, but not as individual components. Beside direct code reuse frameworks also provide knowledge of a certain application domain which can be reused independently from the actual framework code.

The aspect of *Extensibility* communicates best the purpose of frameworks: to provide a basic construct of *Best Practices*, *Design patterns* and *Application Domain logic* on which concrete applications can be build. Concrete extension points are base classes, which can be subclassed, methods which can be overwritten, and interfaces through which external classes can hook into the framework.

The architecture of frameworks often features *Observer* implementations, in which Observables dispatch events to event listeners that have registered to achieve notifications of actions and events. This design style is often called *Inversion of Control*. It is sometimes also called the Hollywood principle: “Don' t call us, we'll call you”. Inversion of control is the main aspect which distinguishes an object-oriented framework from a library, where the

responsibility of application flow stays mostly on the side of the library calling code.

In the following one basic categorization of frameworks will be presented, the categorization into *White-box* and *Black-box* frameworks. The categorization into White-box and Black-box frameworks is not directly related to the J2EE Web tier frameworks, but it is one important and fundamental categorization which helps to understand many architectural issues of framework design.

This categorization is related to a change in the object-oriented programming paradigm. While class hierarchies and sub-classing were the main techniques to build object-oriented programs in the beginning, the paradigm changed with the advance of Design patterns. Especially in the famous *Design Patterns* book [Gamma, 1994] by the 'Gang of Four' two statements question the basic methodologies of traditional object-oriented programming:

Program to an interface, and not to an implementation,

which favors the use of strong interfaces instead of extensive sub-classing,

Favor object composition over inheritance,

which questions the whole subclassing mechanism and suggests a more modular approach as an alternative.

White-box frameworks use the traditional object-oriented approach with inheritance as the basic extension mechanism. *Black-box frameworks* use more component oriented designs where individual components of the framework can be reused through object composition.

2.2. White-box frameworks

White-box reuse of frameworks is defined as using a "software fragment, through its interfaces, while relying on the understanding gained from the study of the actual implementation" [Szyperski, 1997]. Frameworks which rely on inheritance for extension are called White-box frameworks, because it is impossible to use them without prior study of their internal structure.

In cases where multiple components interact with each other in a rather complicated manner it is sometimes necessary to a use a White-box framework instead of a Black-box framework. In these cases the runtime architecture of the

framework can be described on a high level while concentrating on the extension points in the documentation [vanGurp, 2001].

The following examples in Figure 2.1 illustrate the use of a White-box framework. Subclassing of a framework class from an external context, in this case SpecEatingCreature as a subclass of DefaultCreature, represents the White-box reuse of frameworks.

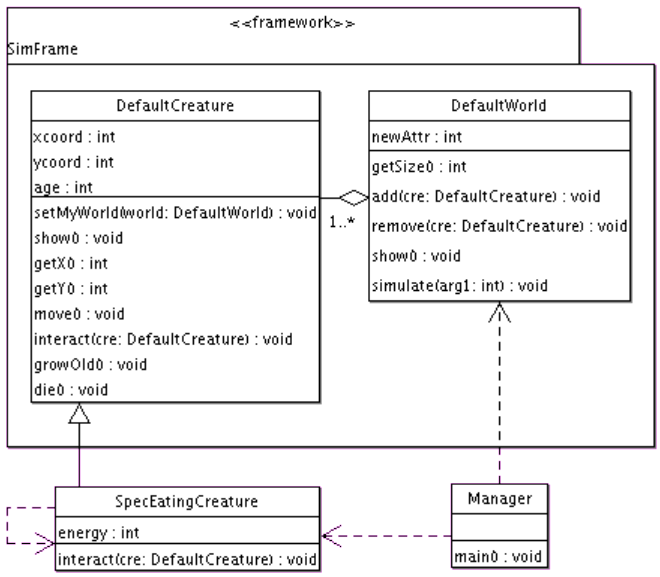

Figure 2.1: Utilization of a White-Box framework [Koskimies, 2003]

2.3. Black-box frameworks

Black-box reuse of frameworks is defined as the "concept of reusing implementations without relying on anything but their interfaces and specifications" [Szyperski, 1997]. Frameworks that can be used by configuring existing components are called Black-box frameworks. Black-box reuse is an ideal rarely achieved.

Because the reuse of Black-box frameworks is limited to the functionality already implemented in the components, the practice of Black-box reuse is seldom used in pure fashion. More common are mixtures of White-box and Black-box reuse. The *White-box layer* of these frameworks covers common aspects of the components and abstract interfaces on which the components are built on. The *Black-box layer* consists of concrete classes and components. Those components can directly be reused, while still preserving the possibility to extend the base classes of the framework [vanGurp, 2001].

The next example in Figure 2.2 shows the usage of a Black-box framework. Composition and usage relations are favoured over static inheritance relationships as only framework classes are instantiated.

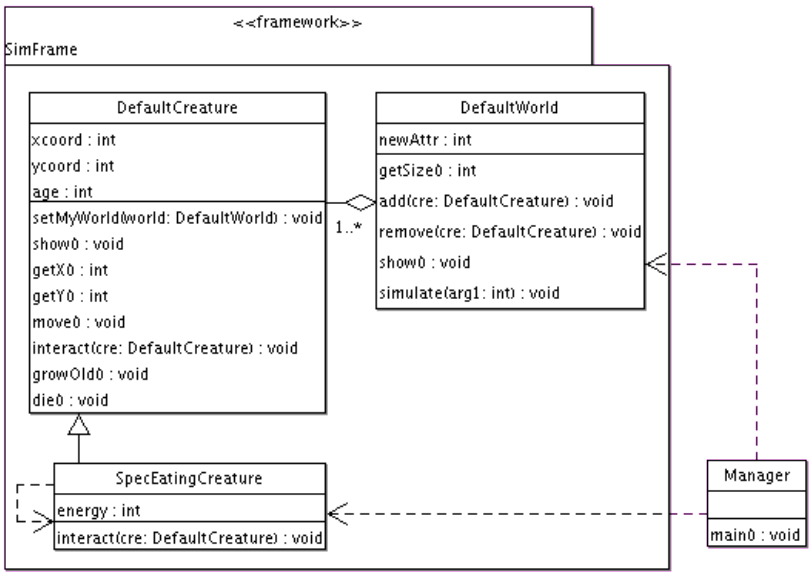

Figure 2.2: Utilization of a Black-Box framework [Koskimies, 2003]

3. Introduction to the J2EE Web tier

In the following the *J2EE Application Model*, the Servlet framework and the *J2EE BluePrints* will be presented. The focus will be on the Servlet framework as it is one of the core elements of this Thesis.

3.1. General Application Model

The general application model of the Enterprise Edition is based on a Multi-Tier-Architecture. The different tiers are the *Client Tier*, the *Middle Tier* with the *Presentation* and *Business Tier* as sub-tiers and finally the *Data Tier*. In the following the functionalities of the tiers will be presented in detail. [Sun J2EEModel, 2004]

Figure 3.1 representing the Java 2 Enterprise Edition emphasizes the division into containers. The Applet and Client Container cover the *Client Tier*, the Web Container (Servlet Container) covers the *Presentation Tier* and the EJB Container the *Business Tier*.

Both the division into containers and tiers describe a logical division of the J2EE architecture. While the container model focuses more on the execution context the tier model describes more a division of architectural concerns (user interface, application, ...).

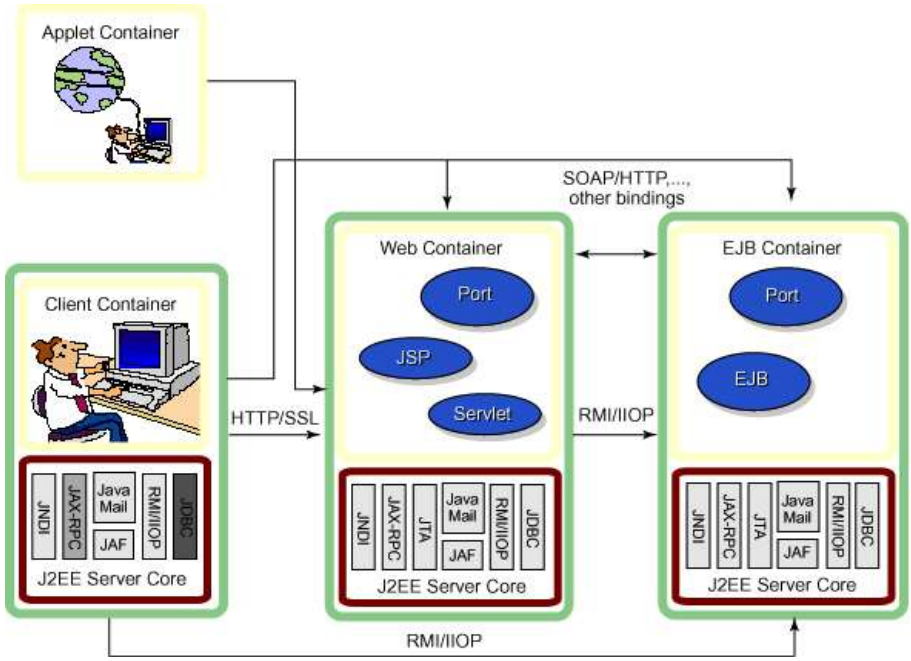

Figure 3.1: *J2EE Container Model [Takawagi, 2001]*

The *Client Tier* is often represented by a web browser, although also other client devices such as PDAs (Personal Digital Assistants), cell phones and desktop applications can be clients of J2EE applications. It is only an implied tier of the J2EE Application model as it is not modelled in the architecture.

The *Web* or *Presentation Tier* is responsible for the rendering of HTML pages and other multimedia elements and delivering them to the browser. Technically this is achieved through a Servlet Container which is bound to a web server. Each application (or service) is represented by a Servlet, which is executed inside the Servlet Container, and responds to HTTP requests with an appropriate HTTP response.

The *Business Tier* is optional for small-scale applications. In cases where legacy systems have to be integrated, this tier covers most of the application logic. The logic is often implemented in Enterprise Java Beans, which come in two basic flavours, session beans and entity beans. *Session beans* model general application logic and often do not contain any state information. *Entity beans* model entities in the business domain and normally contain complex state information.

The state information contained in entity beans often represents the contents of a row in a relational database. The Object-Relational mapping can be managed by the bean itself (bean-managed persistence or BMP) or by the container (container-managed persistence or CMP). Other services the container provides are the automatic creation of proxy objects for client use (mostly for Servlet Containers) and transactional security for communication with databases and legacy systems.

The *Data Tier* which represents Enterprise Information Systems (EIS) and Database Management Systems (RDBMS) is connected to the Business Tier via the Connection and Database APIs of the J2EE.

3.2. Servlet Framework

The Servlet framework is one of the core elements of the Java 2 Enterprise Edition. Originally designed as a general extension framework for Java based server applications, it is nowadays mainly used for Web application development.

The main purpose of the Servlet framework is to provide a standard way to interface a web server with Java applications. Thus it can be seen as a Java variant of the *Common Gateway Interface* (CGI). The main differences are that the Servlet framework is entirely object-oriented and that Servlets are run inside

the Servlet Container as threads and not as separate processes like CGI applications. This results in better interaction between the Servlet Container and the Servlets, faster execution and less memory usage.

3.2.1. JavaServer Pages

In June 1999 Sun announced a new way of using Servlets, the JavaServer Pages (JSP). This was a reaction to the fact that mixing conditionals and text output resulted in Servlets having very ugly code, because each text line had to be printed with a separate *out.println()* command. JavaServer Pages also allow faster deployment as they are compiled directly before usage. Again this was inspired by previous non-Java technologies such as *Active Server Pages* (ASP) by Microsoft and *PHP*.

The JavaServer Pages were one of the main reasons for the growing popularity of Java web development. They allowed programmers to use all the advantages of Java while still preserving a traditional way of web application development and deployment. One of the first drafts of the JavaServer Pages Specifications presented two models how JavaServer Pages could effectively be deployed in Web applications [Sun JavaServerPages, 1998]. In later versions of the specifications, the intuitive division into two basic models was replaced by palettes of more technical usage scenarios. The two initial models will be presented in the following.

3.2.2. Model 1

In the Model 1 pattern, which is illustrated in Figure 3.2, a *Designer / Developer separation* was promoted by putting all application logic into Java Beans that were accessed from within the Java Server Pages. The code parts which were left in the Java Server Pages were responsible for visual formatting and interweaving static HTML elements with output from session attributes and application logic.

One of the problems this pattern didn't solve was the separation of *Page flow*, *Error handling* and *Form validation* mechanisms from the formatting code inside the Java Server Pages.

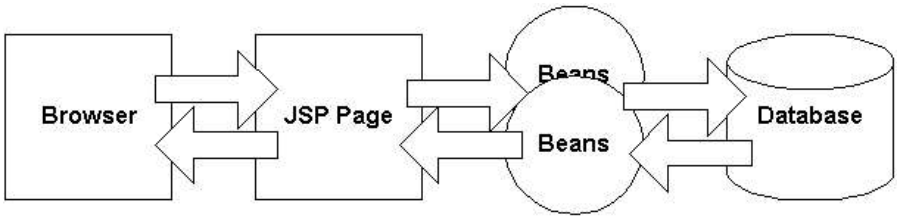

Figure 3.2: *Model 1*

3.2.3. Model 2

The second model was the *Model 2* pattern which is illustrated in Figure 3.3. Using this pattern the separation of concerns went further. Servlets and Java Server Pages were no longer directly bound to URLs (Universal Resource Location), but were controlled by a *Central Dispatching* mechanism.

All requests are initially fed into a central Servlet which populates Java Beans based on request parameters and dispatches the control to JavaServer Pages for the rendering of HTML responses. Most of the application logic is now separate from the rendering code of the Java Server Pages.

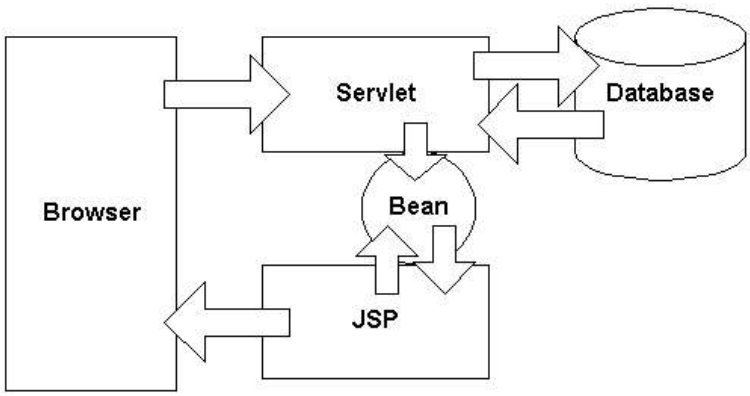

Figure 3.3: *Model 2*

3.2.4. JavaServer Pages Standard Tag Library

The JavaServer Pages Standard Tag Library is a way to replace common repetitive Java Code such as Java Bean lookup, conditionals, loops and XML processing by shorter XML tag equivalents. The goal of JSTL is to simplify JavaServer Pages development by providing most of the general functionality in XML equivalents.

The final release of the JSTL specification was released in June 2002 and was quickly adopted by JavaServer Pages developers [Sun JSTL, 2003]. As many HTML designers are not familiar with the Java language, an XML scripting interface to the most common commands is preferable to the

combination of clumsy chunks of Java Code mixed with otherwise declarative (X)HTML code.

The following code example show an excerpt from a JSP page, where a Java Bean is accessed, looped through and its items are displayed in an HTML table. The first variant (1) is done using standard JSP techniques, whereas the second one (2) uses JSTL [Bruchez BothWorlds, 2003].

(1) without JSTL

```
<%
  MyTableData tableData = MyDAO.queryData();
%>
<table border="1">
  <%
    for (int i = 0; i < tableData.size(); i++) {
      String cellColor = (i % 2 == 0) ? "gray" : "white";
  %>
  <tr>
    <td bgcolor="<%= cellColor %>">
      <%= i %>
    </td>
...
  <% }%>
</table>
```

(2) with JSTL

```
<table border="1">
<c:forEach items="${tableData.data}"
   var="currentData"
   varStatus="status">

  <c:set var="cellColor">
    <c:choose>
      <c:when test="${status.index % 2 == 0}">
        gray
      </c:when>
      <c:otherwise>white</c:otherwise>
    </c:choose>
```

```
  </c:set>

  <tr>
    <td bgcolor="${cellColor}">
      ${status.index}
    </td>
   ...
  </c:forEach>
</table>
```

In addition to providing a declarative XML interface, a scripting like expression language is included in the JSTL to make conditionals and looping more compact. The standard functionality of the included tag libraries cover accessing URL-based resources, Internationalization (i18n) and text formatting, Relational database access (SQL), XML processing and String manipulation.

3.3. J2EE BluePrints

The J2EE BluePrints are a set of design patterns for the different tiers in the J2EE Platform, which are illustrated in Figure 3.4. Originally published as a guidebook with guidelines, patterns and code samples, the book title became quickly a synonym for the presented design patterns.

The basic categorization is by the tiers presented in the J2EE application model. Most of the patterns are based on best practices to reduce *Code duplication*, *Network traffic*, *Class dependencies* and *System complexity*.

In the following I will briefly present the Web Tier patterns of the J2EE Blue Prints with a special focus on the Front Controller pattern, as it is the basic design pattern behind most of the Web Tier frameworks. [Sun J2EEPatterns, 2002]

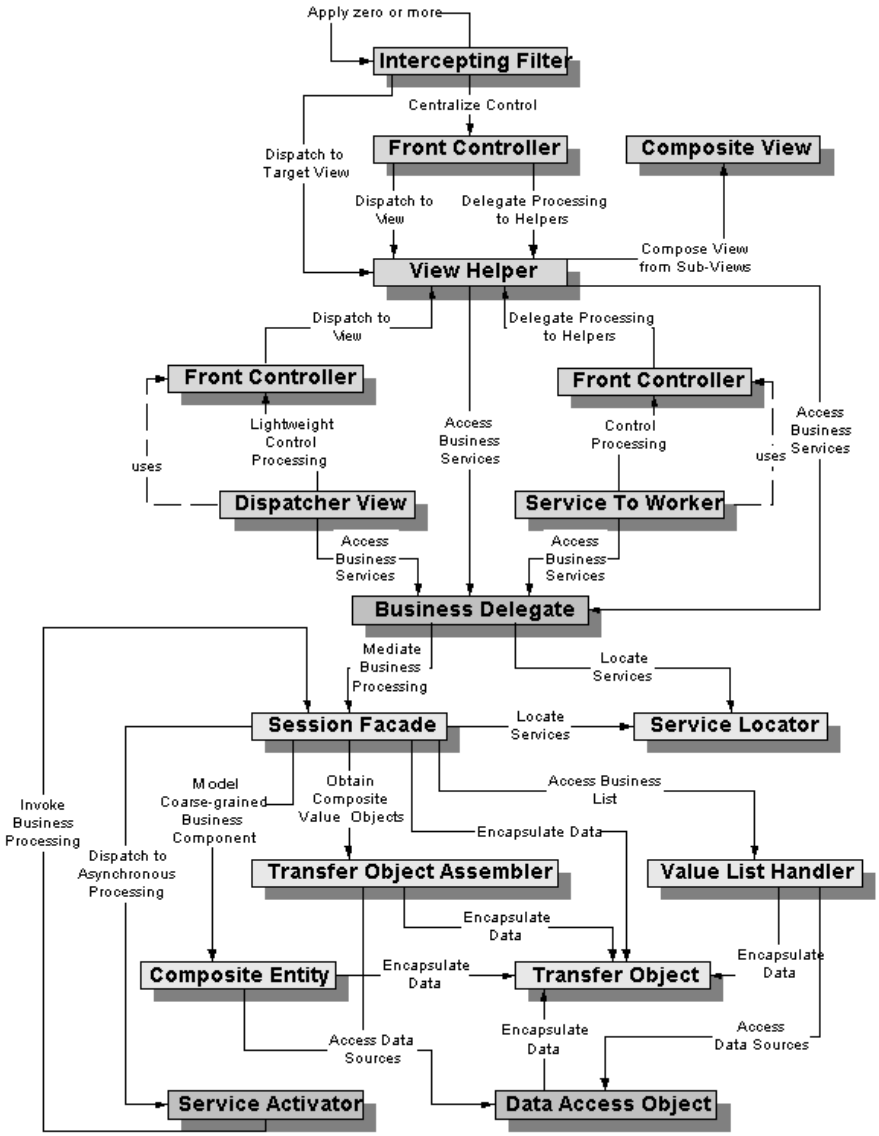

Figure 3.4: *J2EE Blue Prints*

3.3.1. Front Controller

The *Front Controller* (Figure 3.5) provides a central entry point for request processing and view dispatching, which is often implemented as a Servlet. The main tasks of the controller are invoking *Security services*, delegating *Business processing*, *View dispatching* and *Error handling*. The centralization of these aspects reduces code duplication in the JavaServer Pages code and promotes a separation of request processing logic from view creation logic.

The *Front Controller* pattern takes its name from the *Model-View-Controller* pattern, a standard GUI design pattern originally used in the Smalltalk environment and later also in the Java Swing framework. The Model-View-Controller defines the responsibilities of User Interface components (widgets as well as more complex components) into three different elements: the *Model*, which contains the state information of the component, the *View*, which is

responsible of the visual representation and the *Controller*, which mediates between the *Model* and the *View*.

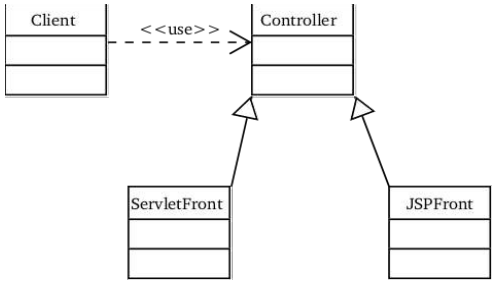

Figure 3.5: *Front Controller*

3.3.2. Intercepting Filter

Intercepting Filters (Figure 3.6) are pluggable elements inside the request processing chain of the Servlet Container. They can perform both pre-processing of incoming requests as well as post-processing of out-going responses. The main implementation techniques are internal functions inside the *Front Controller* and *Servlet Filters*.

Servlet Filters are declared in the Web Application deployment descriptors of the Servlet containers and cover mostly common filtering tasks such as *Authentication*, *Authorization*, *Logging* and *Debugging*. Filters and Filter Chains can be mapped to certain URL patterns in the same way as Servlets.

The main purpose of the *Intercepting Filters* is to separate request filtering logic from request processing logic and to provide an easy way to write reusable components for the filtering tasks. On a more abstract level, the Intercepting Filter pattern can be seen as the application of the *Decorator* pattern to the J2EE Web Tier. The Decorator pattern adds services to an existing component or class by wrapping or subclassing its interface and introducing new methods and extending old ones.

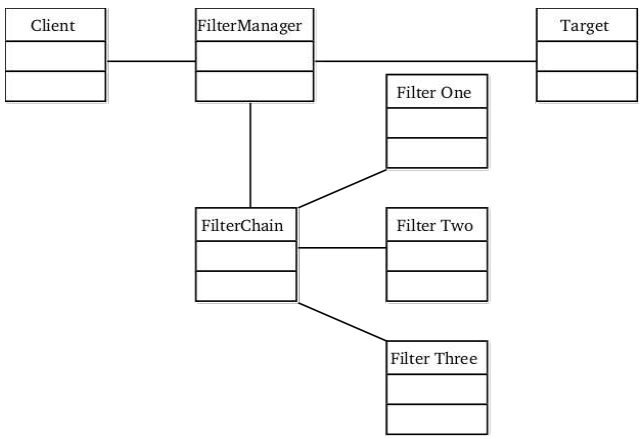

Figure 3.6: *Intercepting Filter*

3.3.3. Composite View

The *Composite View* pattern (Figure 3.7) is a structural pattern that promotes a modular design of the view elements. A composite view consists of multiple atomic subviews that are integrated into a full view. This pattern is especially useful, if the user interface has lots of dynamic and static view elements, if the service is a portal with lots of independent subviews or if certain graphical elements (header, logo, footer) should be separated from the dynamic view elements.

This pattern can be seen as a direct application of the *Composite* pattern to JavaServer Pages and other view technologies. Commonly the Composite pattern describes a hierarchical class structure, where the container and component classes are derived from the same super class. In this case the direct implementation can be in form of file includes, the Jakarta Tiles technology, an abstraction of file includes, or more lightweight technologies such as templates inside an XSL-T stylesheet.

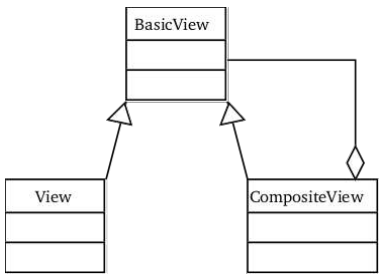

Figure 3.7: *Composite View*

3.3.4. View Helper

The *View Helper* pattern (Figure 3.8) is also closely related to the *Model-View-Controller* pattern. While view components are commonly used to represent particular business requests, more complex processing and formatting responsibilities should be delegated to helper classes. In scenarios where JavaServer Pages are used as the view components, the helper classes are often implemented as JavaBeans or custom tags. Helper classes can often be used for multiple views and thus promote code reuse and enhance maintainability.

View Helpers can also be used as *Adapters* to the data models, in cases where the model is too complex to be accessed directly from the View components. The Adapter pattern describes a scenario where an external fixed

interface is glued to an internal interface of a component with the same purpose. Typical examples of such scenarios are the use of JDBC for interaction with relational databases, use of Enterprise Java Beans or direct access to complex proprietary data models from the Web Tier.

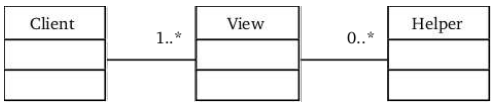

Figure 3.8: *View Helper*

3.3.5. Dispatcher View

The *Dispatcher View* pattern (Figure 3.9) is closely related to the *Front Controller* pattern. A Dispatcher is used in cases where view and navigation management should be separated from the view components. In this design pattern the content retrieval and business logic execution is deferred to the time of view processing.

It should be used in cases where control flow is relatively simple and mostly based on parameters of the request and not on results of business logic or content retrieval. A Dispatcher can be implemented as a part of the *Front Controller*, as a separate component or as helper class in the View. In the *Dispatcher View* pattern the *Front Controller* is a relatively simple component, which only takes care of security checks and view dispatching.

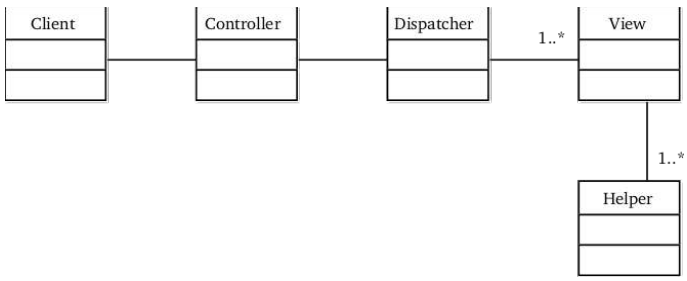

Figure 3.9: *Dispatcher View*

3.3.6. Service To Worker

The *Service To Worker* pattern (Figure 3.10) is similar to the *Dispatcher View* pattern, but places content retrieval and business logic execution into the *Front Controller* component. In this pattern the Controller dispatches the request processing to worker classes, which delegate the requests further to business objects or execute the business logic themselves. Based on the results of the

business logic or content retrieval, a suitable view component is chosen to generate a response.

The *Service To Worker* pattern is not equivalent to a certain component, but describes a combination of the *Front Controller*, Worker classes and a Dispatcher component.

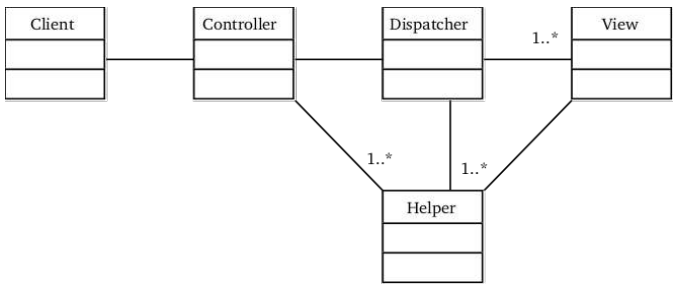

Figure 3.10: *Service To Worker*

3.3.7. Business Delegate

The *Business Delegate* pattern (Figure 3.11) presents a similar *Adapter* solution like the *View Helper* pattern, but instead of just providing abstractions of single services, a complete remote business service is hidden behind the *Business Delegate*. This pattern is useful for cases where the business system is implemented in Enterprise Java Beans, because the lookup and access of the remote services can be quite complex in these cases.

The main tasks of the *Business Delegate* are centralizing the access to remote services, aggregating service calls, caching the results and converting service level exceptions into application level exceptions. If the *Business Delegate* is designed without dependencies on the Web Tier framework, it can also be used for other GUI solutions such as a Swing GUI.

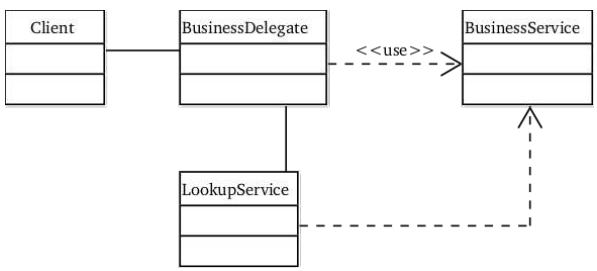

Figure 3.11: *Business Delegate*

3.4. J2EE Web tier frameworks

The function of the J2EE Web Tier frameworks can only be understood in relation to the Servlet framework and its limitations. While the Servlet frameworks represent just a basic *Client/Server model*, the more specific Web tier frameworks are representations of *Web Application models*.

The web application models represented by the frameworks of this thesis are *Portals* (Portlet Specifications), *Publication channels* (Apache Cocoon), *Model 2* like applications (Jakarta Struts) and *Desktop GUIs* (JavaServer Faces). Beside the promotion of certain application models the presented frameworks provide also support for more general application design concerns such as Navigation, Form Validation, Error handling, Internationalization and Security (Authentication and Authorization).

More specific issues are the integration of other Template mechanisms beside the Java Server Pages such as WebMacro, Velocity etc., enhanced support for the Designer / Developer separation, extensions of the Servlet Security functionality, Error Handling independent from view presentation etc. [Hunter, 2002].

4. The approach and its advantages

In this part I will describe my first steps in writing this thesis, the final methodological basis and introduce the J2EE Web tier frameworks, the main elements of my analysis.

4.1. Sun and Apache Frameworks

The frameworks I will present in this paper are *Jakarta Struts* and *Cocoon* from the Apache Software Foundation and the *JavaServer Faces* (JSF) and *Portlet* Specification from Sun Microsystems. All of the chosen frameworks enhance the Servlet framework with different aspects.

For each framework I will give a short introduction, present the main architectural features of the major releases and compare them with the Servlet framework. Additionally main application design issues and their support in the different frameworks will be presented.

I chose frameworks from Sun and Apache, because they are currently the major players in Java Web Application domain. In fact Apache even provides some reference implementations of J2EE technologies such as the Jakarta Tomcat Servlet container, the Jakarta Struts framework, the Xerces XML parser and the Xalan XSL stylesheet processor.

4.1.1. Jakarta Struts

Jakarta Struts can be seen as the reference implementation of the J2EE Web Tier BluePrints and is probably the most popular Java web application framework. Jakarta Struts was originally created by *Craig McClanahan* and was donated to the Apache Software Foundation in 2000. It provides a thin, but flexible framework around a variation of the Model-View-Controller design pattern.

4.1.2. JavaServer Faces Specification

JavaServer Faces is a component framework for reusable UI components that facilitate the development of Java web applications. It is an attempt to enhance the Servlet technology with an additional component framework that addresses similar issues like the Jakarta Struts framework. *Craig McClanahan*, the main architect behind the Jakarta Struts framework, is also one of the editors of JavaServer Faces Specification.

The main features the JavaServer Faces framework provides are the management of user interface component state across requests, form

processing, enhancements of the Servlet event model, validation of request data and page-to-page navigation [Sun JavaServerFaces, 2003].

4.1.3. Apache Cocoon

Apache Cocoon is a general XML processing and publication engine, but is mostly used in combination with a Servlet connector as a web application framework with focus on aggregating and formatting heterogeneous XML content. Originally designed in 1998 by *Jon Stevens* and *Stefano Mazzocchi* as a simple Servlet to manage the automatic page formatting of the java.apache.org site, the system became quickly one of the most popular XML publishing engines for the J2EE platform [Mazzocchi, 2003].

4.1.4. Java Portlet Specification

The Sun ONE Portal Server and the related Portlet Specification are technologies that were designed to facilitate and standardize the development of portals on the J2EE platform. A portal can generally be described as a web application that provides *Customization*, *Aggregation* and *Presentation* of services from different sources such as Web Services, general Enterprise Information Systems and other services.

The Portlet Specification is based on the Servlet Specification v2.3 [Sun Servlet, 2001] as both component technologies address common aspects. The main extensions a the Portlet Specification provides to the Servlet framework are access to persistent configuration and customization data, access to user profile information, URL rewriting functions and enhanced session management [Sun JavaPortlet, 2003].

5. Jakarta Struts

Jakarta Struts (Figure 5.1) is one of the most popular Web Tier frameworks for the Java platform. The most important reason for its popularity is probably the architectural simplicity and its tight integration into the J2EE platform. It is mainly based on the *Model 2* pattern and focuses on the Controller component, but integrates well with standard Model and View technologies.

Jakarta Struts has direct support for standard data access technologies such as JDBC and EJB, as well as some Object-Relational mappings to implement the Model part. For the Presentation part Struts integrates well with JavaServer Pages, including the Standard Tag Libraries and JavaServer Faces, as well as Jakarta Velocity, XSLT, and other presentation technologies. [JakartaStruts, 2003]

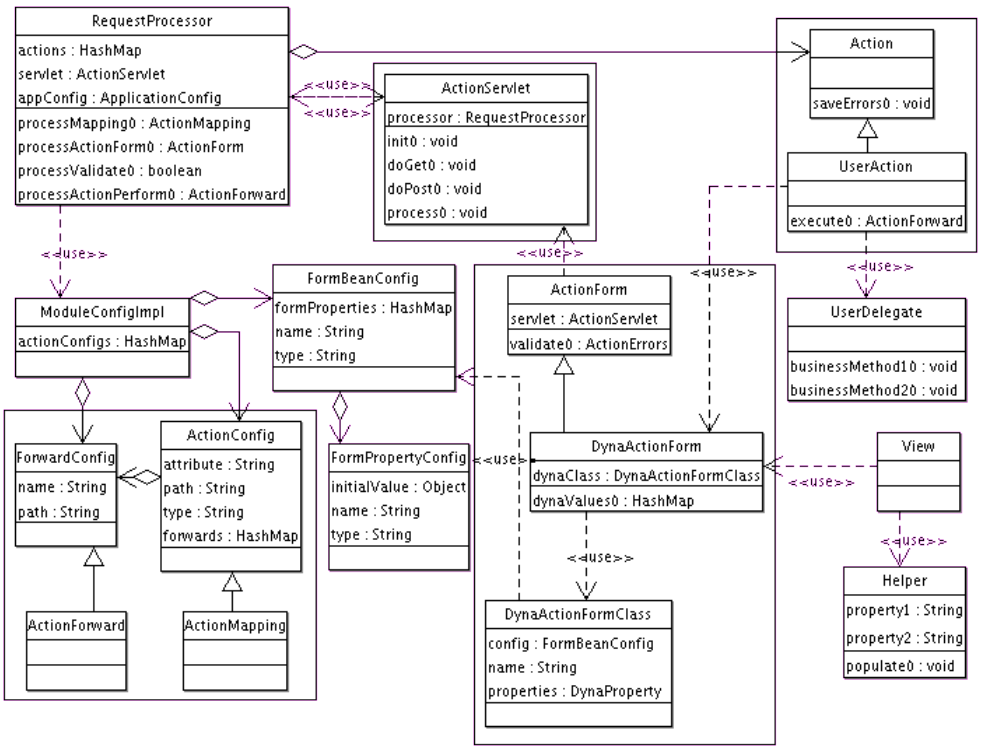

Figure 5.1: *Jakarta Struts – UML class diagram*

5.1. Architecture

Instead of focusing on runtime behaviour of the Jakarta Struts framework, I will present the main classes and components of the framework in the design pattern contexts, in which they appear. In the following, each of the pattern implementations will be presented in detail. The components of the framework are mainly based on four design patterns of the J2EE BluePrints pattern

collection: the *Front Controller* pattern, the *View Helper* pattern, the *Dispatcher View* pattern and the *Service To Worker* pattern.

5.1.1. Front Controller

The *Front Controller* implementation in Jakarta Struts consists mainly of the *ActionServlet* and its related *Request Processor*. The main tasks of the ActionServlet are to invoke Actions and ActionForwards based on the URL and other request parameters, to populate Form Beans that are used in the Actions and Views and to handle Internationalization and Content type issues.

The ActionServlet is configured in an XML file (struts-config.xml) in a similar fashion like an ordinary Web application under a J2EE compliant Servlet container. The resources that are configured in this XML file are *ActionMappings* that map URL patterns to Actions, ActionForwards, that define the output pages, and ActionForms, which declare the HTML forms and the related validation logic.

Actions, ActionForwards and ActionForms are declared in the ActionMappings with logical names that refer to their purpose inside the application scope instead of Java class names. This makes the configuration and maintenance of applications much easier and is especially useful in the case of JavaServer Pages, which can't be accessed directly if used inside a Struts based web application.

5.1.2. Service To Worker

The mentioned *Action* classes represent the *Service To Worker* pattern in Jakarta Struts. Actions are bound to certain URL patterns and are invoked by the Request Processor to process HTTP requests and to make changes to the Business Model. For each URL pattern one Action instance is created for the ActionServlet.

The result of the Action execution is an ActionForward that defines to which View or Action object the control should be forwarded after the execution. Examples of common usage scenarios of Action objects will be presented in the following.

The validation of the current session state is often delegated to an Action object. If the Action object finds that the user is not logged in, the request can be forwarded to a login or presentation page from which the user can log into the application.

Another common usage of Action objects is to validate form data. The Form Bean properties will be validated and for each error an appropriate error message key will be appended to the request as an attribute. If the validation fails the control can be forwarded back to the input form. More generally formulated, Actions can be used to populate and manipulate server-side objects inside the session or request scope, which are then used by View components.

The most common usage of Actions is to perform some operations on a relational database such as inserting or updating a row with Form Bean data or querying the database. For more complex applications where application logic should be separated from the business logic, the database operations can be specified in Delegate and Helper classes which are called from within an Action.

5.1.3. Dispatcher View

The *Dispatcher View* pattern in Jakarta Struts defines abstract references to *Action* and *ActionForward* objects in the XML configuration file. Common Java Bean reflection techniques are used to populate the ActionForward and Action instances from the XML configuration file. While the use of abstract view names is more or less still optional in Struts 1.0, it is obligatory in Struts 1.1, where the direct access to JavaServer Pages is no longer possible. This policy enforces a very strict use of the Model 2 pattern.

Another implemented pattern which is related to both the *Dispatcher View* and the *View Helper* pattern is the *Composite View* pattern. Because direct access of View components can be considered as an antipattern in this context, the Struts framework promotes alternatives to the direct inclusion of JavaServer Pages to form composite views.

Tiles is the name for this technique in the Jakarta Struts. The Tiles framework is not much more than a dispatching mechanism to include common static or dynamic page elements into JavaServer Pages to enforce a common look-and-feel for different pages inside a web application. More concrete application scenarios of the Tiles framework will be presented later.

5.1.4. View Helper

The *ActionForm* can be considered as an implementation of the *View Helper* in the Jakarta Struts framework. It serves as a mediator between the Action and the View classes. ActionForms are used as object representation of HTML form inputs and form sequences. They are declared in the Struts configuration file

and are mapped to URL patterns in the same way as Actions and ActionForwards. The ActionServlet creates and populates the ActionForm objects before the invocation of the Actions when necessary.

The purpose of using ActionForms instead of simple Java Beans is to enforce a certain usage of mediator elements and to make the declaration of HTML forms more explicit. ActionForms facilitate form validation and can directly be defined in the configuration file in newer versions of Jakarta Struts.

Jakarta Struts also enhances the standard JavaServer Pages Tag Libraries with its own additions. The most often used are probably the *HTML Tags* which offer compact XML tags for input fields and buttons that can directly be bound to the properties of Form Beans. The *Bean Taglib* library contains tags that can be used to define new beans in any desired scope as well as tags to render the content of a particular bean to the output response. The *Logic Taglib* library contains tags to manage conditional generation of output text, looping over object collections and application flow management.

In general the tags defined in the JavaServer Pages Standard Tag Library (JSTL) should be favoured over the Struts tags as they are compatible with the J2EE standards and can also be used in other JavaServer Pages contexts.

5.2. Concerns

In the following the main concerns of the Jakarta Struts framework will be addressed. The concerns discussed are navigation, form validation, error handling, internationalization and security. The same pattern will also be used in subsequent sections.

5.2.1. Navigation

In Jakarta Struts Navigation is addressed in a rather sophisticated way. The URL namespace used by the Struts Servlet uses abstract entry points that are either directly dispatched to correspondent views or forwarded to Actions.

In this way the package namespace of the Actions, the directory namespace of the JavaServer Pages and the URL namespace are kept separate to a very high degree. Instead of declaring each mapping individually wildcard mappings can be used in Struts 2.0 to group similar mappings into generic declarations.

While the namespaces of actions, views and the URL addresses are kept separate in a very clean way, there' sno support for navigation on the view

level except for some JSP Taglib shortcuts. Such support can be added through the JavaServer Faces which can be integrated into Struts via JSP Taglibs.

5.2.2. Form Validation

Forms in Jakarta Struts are modelled as *ActionForms* and *DynaActionForms*. The related validation can either be done by overriding a "stub" method in the ActionForm or by delegating the validation to an Action. If the former approach is chosen the validation is performed in the validate method. If the validation succeeds, the method returns null or an empty ActionErrors instance. If the validation fails, an ActionErrors instance with appropriate error message mappings is returned.

Validation inside the ActionForm can also be delegated to the *Struts Validator*, which acts as plug-in to provide automatic validation functionality to Forms. ActionForms that should use the validation of the Struts validator must extend ValidatorForm. Forms and their related validations are declared in an XML file in a similar way like Actions and ActionMappings.

5.2.3. Error Handling

While exception catching on a reasonably small scope is advised programming practice, this is nearly impossible for applications that are executed in distributed environments. For this reason Jakarta Struts offers a means to define general *ExceptionHandlers* that catch Exceptions thrown by Actions and perform appropriate error handling.

ExceptionHandlers are declared in the struts-config.xml file in the following way.

```
<global-exceptions>
  <exception key="some.key" type="java.io.IOException"
      handler="com.yourcorp.ExceptionHandler"/>
</global-exceptions>
```

The variable key references to a message in the ResourceBundle, type to the class of the Exception to be mapped to this Exception handler and handler to the class of the handler.

5.2.4. Internationalization

For Internationalization issues Jakarta Struts relies completely on the standard Java techniques for providing locale specific displays of error messages as well as date, time and number formats. Detailed descriptions of Internationalization issues will be omitted as they are not Jakarta Struts specific and too complex to be described in short.

5.2.5. Security

Jakarta Struts can use the declarative security facilities of the Servlet or an EJB Container, but includes also support for the standard *Java Authentication and Authorization Service* (JAAS). The issue of web application security is also too broad to be discussed in the context of the paper and will this be omitted.

5.3. Evolution

The changes between the first version of Jakarta Struts (v. 1.0) and the second one (v. 1.1) are not as substantial as in other frameworks (e.g. Apache Cocoon), but they nevertheless provide insight for understanding into which direction the Jakarta Struts framework is evolving. The most important changes will be presented in the following. [Yu, 2002]

5.3.1. Multiple Sub-applications

One of the main problems with Struts 1.0 was the manageability of the configuration file for larger projects. A Struts-based application has to run under one ActionServlet, and the ActionServlet can use only one configuration file. For small or medium-sized projects this is no problem, but for large projects the maintenance of a single configuration file for all the application mappings is nearly impossible.

The problem is solved in Struts 1.1 by introducing sub-applications. Each sub-application has its own configuration file which can be maintained by a sub team independently from the other configuration files.

This solution resembles the mapping scheme in the servlet context, where the Servlet Container determines the appropriate web application context based on the URI path. Struts 1.1 maps the second prefix of the URI path to a sub-application with an independent configuration file. In cases where Struts 1.0 application has to be deployed in Struts 1.1 context, a default sub-application is

defined which is matched against an empty prefix. Again this is similar to the usage of a default web application context in the Servlet Container.

5.3.2. DynaActionForms

Another major problem in the usage of Struts 1.0 is related to the writing of ActionForms. ActionForms are object oriented representations of HTML forms and wizard-like form sequences. For each HTML form or form sequence an individual ActionForm has to be written.

The properties of the ActionForm are accessed according to the Java Bean standard, which means that for each form property a pair of accessors has to be written. As another architectural constraint the ActionForms are designed in such a way that the reuse of model-layer objects (such as from the Business tier) as ActionForms is nearly impossible. The reason for this is the fact that most of the components of the Jakarta Struts framework are designed as concrete classes instead of interfaces, which reduces possibilities to mix Struts components which Business tier objects and other external components.

DynaActionForms are the alternative offered by Struts 1.1. They are based on DynaBeans, type-safe name-value pairs that can be used in a similar fashion like Java Beans through the Java Reflection API. The main benefit of using DynaActionForms instead of normal ActionForms is that the properties of the former can be declared in the Struts configuration file and are initialized and populated with the given data. The main difference in the usage is that the properties of a DynaActionForm have to be accessed with a generic *getProperty* and *setProperty* pair instead of the general Java Bean accessors.

5.3.3. Declarative Exception Handling

ExceptionHandlers where added in Struts 1.1 to provide a means to configure exception handling policies declaratively in a similar fashion to the declaration of Actions and ActionFowards. ExceptionHandlers can be mapped to specific exception types and are execute whenever an Action throws an exception of the appropriate type. The default ExceptionHandler can be used to forward the control to error pages when uncaught exceptions are thrown by an Action instance. Subclassed ExceptionHandlers can be created to perform additionally more specific operations when exceptions are thrown, such as logging.

The usage of ExceptionHandlers centralizes exception handling in the Struts configuration file and facilitates the development of compact Action

classes which focus on core application logic. Especially when EJBs or JDBC are used, the palette of possible exceptions is quite high.

5.3.4. Validator

The Validator provides a means to define validation rules to validate user input of HTML forms. In addition to creating the related server-side validation code the Validator also creates client-side validation code (i.e. JavaScript) for the same validation rules. The validation rules are based on regular expression pattern matching and some commonly used validators are already shipped with the framework.

The validation rules are declared in a configuration file and can be used when the Form classes are subclasses of ValidatorForm or ValidatorActionForm instead of ActionForm. Validation rules for forms can be grouped and language or country specific variants can be used based on the active Locale.

5.3.5. Integration of JavaServer Faces

The integration of JavaServer Faces is one of the main directions in the Jakarta Struts evolution. As Craig McClanahan, the main developer behind Jakarta Struts, is also one of the lead designers of the JavaServer Faces reference implementation, scenarios where Jakarta Struts and JavaServer Faces are used simultaneously are promoted quite much.

Currently the integration of JavaServer Faces is possible through JSP Taglibs on the View level, but the JavaServer Faces design might also influence some deeper architectural constraints such as ActionMappings, ActionForwards etc.

6. JavaServer Faces Specification

JavaServer Faces is a user interface (UI) component framework for Java web applications which provides a customizable component model for user interface elements, an event handling model which is based on the Java Bean standard (*EventListeners* and *ActionListeners*), a validation framework and pluggable components to customize the rendering of the user interface components.

Similar to Jakarta Struts it provides a variation of the *Model 2* pattern for Java Web application development, but instead of providing a thin controller layer and some helpers to facilitate form validation and error handling it provides a component framework for user interface elements, which tries to hide away most of the complex aspects of web application design.

The model thus becomes more similar to a desktop *GUI framework* such as Java Swing. The component framework resembles the Java Swing framework in many aspects such as the use of the Composite View pattern, the Java Bean based Observer pattern and pluggable Look-and-Feel. While a JFrame or Window acts as a main container for user interface components in a Swing application, a JSP page or Velocity template acts as a main container for components of the JavaServer Faces framework.

6.1. Architecture

The architectural overview of the JavaServer Faces won't be based on any standard design pattern set. Instead architectural concerns and components will be used as structuring elements. The presented issues are the *User interface component model*, the *Event handling model*, the *Validation framework* and the *Rendering model*.

6.1.1. User interface component model

The user interface components are the basic elements from which a JSF based web user interface is built. The elements can represent single elements of an HTML form such as text fields, selection boxes or lists and can optionally be associated to elements of the business model via value reference expressions.

As the components can be hierarchically organized, they can also be used in more complex cases to render a calendar component in a portal, an entire HTML table or any other complex element that can easily be externalized from the structural page formatting code.

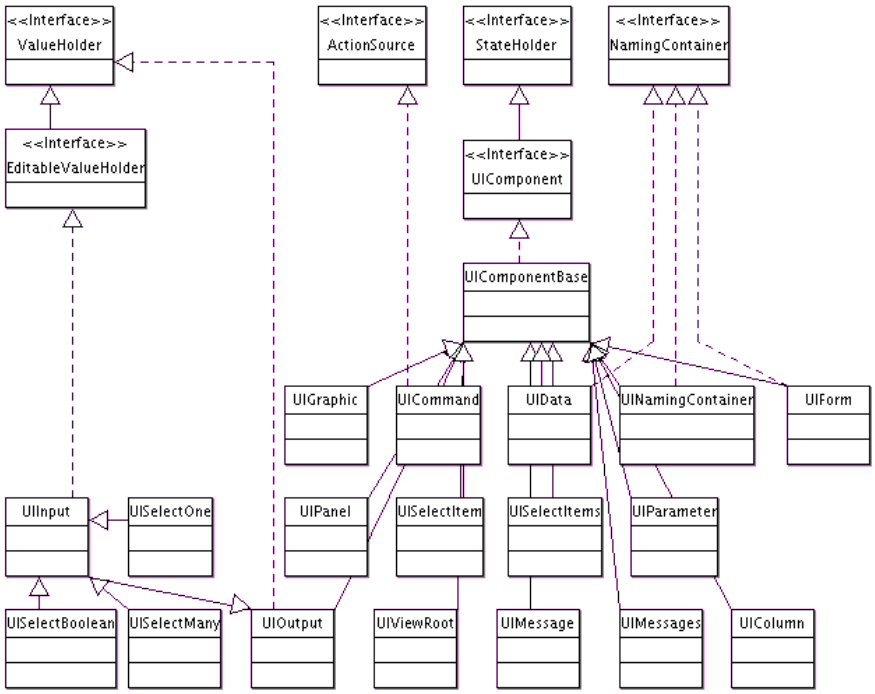

Figure 6.1: *Class diagram of the JavaServer Faces UI components*

To control navigation flow, validate user input and to support different kinds of mark-up creation various helper APIs are bound to the user interfaces. *Converters* are used to provide pluggable support for direct conversion to and from object types from the business model to the XML mark-up used in the presentation. *Events and Listeners* are based on the patterns of the Java Bean specification and provide mechanisms to attach application actions to user interface events. The page navigation is also based on this technique. Finally *Validators* offer pluggable functionality to check the validity of the input for the user interface components. The alternations to page flow and the displaying of error messages are automatically handled.

Components are arranged hierarchically in component tress. Beside a normal parent-child relationship components can also be attached to parents as facets. This can be especially useful in cases where the parent component represents an HTML table, whose children are rows in that table and the facets decorating elements such as footer or header components.

6.1.2. Event handling model

The event handling model of JSF is based on the design patterns of the Java Bean specification. Like the basic component model also this aspect is quite similar to the way the Swing framework handles events and event notifications.

UIComponent subclasses emit events to registered listeners to signify internal state changes. The events are cued by the JSF implementation and are forwarded to the registered listeners at the end of certain request processing phases.

Events that are broadcast by a UIComponent must extend the *FacesEvent* class. The basic subclasses are the *ValueChangeEvent* that notifies registered listeners of value changes and *ActionEvent* which represents a user interface action such as the submission of an HTML form.

On the Listener side the *FacesListener* interface acts as the main interface. The *ActionListener* and *ValueChangedListener* interfaces are bound to the ActionEvents and ValueChangedEvents respectively. ActionSources such as Buttons or Links can emit ActionEvents whereas UIInputs, components of HTML forms, emit ValueChangedEvents. More generally UIComponents emit FacesEvents.

6.1.3. Validation framework

As previously mentioned JavaServer Faces provides means to attach validation functionality to user interface components. Zero or more validators can be attached on each UIInput component. During the Process Validations phase of the request processing lifecycle validators perform checks on the local values of the component and force a redisplay of the input form if any of the validations fail.

Validators can be attached to individual components through JSTL tags or can directly be integrated into the components. Compared to Jakarta Struts the validation mechanisms are more fine-grained as they are bound to individual components whereas they are related to complete ActionForms in Jakarta Struts. The advantage of binding validation to complete forms is that the output of the validation might be dependent on certain value combinations instead of individual values.

6.1.4. Rendering model

JavaServer Faces features two models for decoding component values from incoming requests and encoding component values into outgoing responses. In the *Direct Implementation* model the components have to encode and decode their values themselves. In the *Delegated Implementation* model the decoding and encoding of the values is delegated to Renderers. This design approach separates the content of the component (the Model) from its representation (the View). *RenderKits* combine Renderers of different component types to form

pluggable Look-and-Feels. RenderKits are useful to customize the rendering for different mark-up types such as XHTML, more general XML variations and WML.

The separation into a component which acts as basic data container and delegated rendering is a variation of the *Model-View-Controller* pattern on a more fine-grained level. This design approach is similar to the rendering mechanism of the Swing framework, where the rendering of widgets and other GUI elements is delegated to the Look-and-Feel implementation currently activated in the application scope.

6.2. Concerns

The same categorization of concerns as in the previous chapter will be used, although the coverage is not as broad as in the Jakarta Struts, as the JavaServer Faces framework focuses on the user interface aspects of web applications. The focus is therefore on *Navigation* and *Form Validation*.

6.2.1. Navigation

Navigation in the URL space of the application is handled by a single *NavigationHandler*. The subclassing the NavigationHandler and overwriting a template method, the navigation handling can be defined procedurally. The preferred way though is to use the default NavigationHandler to define the navigation handling declaratively. The behaviour of the default NavigationHandler is configured from the contents of zero or more XML configurations files. The syntax and semantics of these declarations will be presented in the following.

On the top level the navigation handling is described by zero or more <navigation-rule/> elements, which are matched to the request via wildcard expression. The match is expressed by a <from-view-id/> element. Inside the <navigation-rule/> element are multiple <navigation-case/> which describe the actual navigation cases. A navigation case is either matched by the outcome of application action or the expression referencing the action. The <navigation-case/> element includes a <to-view-id/> whose content is the view identifier that will be selected.

In the following an excerpt of an example Navigation Handler configuration will be presented. Explanations of each core element are included in <description/> elements.

```
<navigation-rule>
```

```
  <description>
    APPLICATION WIDE NAVIGATION HANDLING
  </description>
  <from-view-id> * </from-view-id>

  <navigation-case>
    <description>
      Assume there is a “Logout” button on every page that
      invokes the logout Action.
    </description>

    <display-name>Generic Logout Button</display-name>
    <from-action>#{userBean.logout}</from-action>
    <to-view-id>/logout.jsp</to-view-id>
  </navigation-case>

  <navigation-case>
    <description>
      Handle a generic error outcome that might be returned
      by any application Action.
    </description>

    <display-name>Generic Error Outcome</display-name>
    <from-outcome>loginRequired</from-outcome>
    <to-view-id>/must-login-first.jsp</to-view-id>
  </navigation-case>

</navigation-rule>
```

6.2.2. Form Validation

The support for form validation in JavaServer Faces is limited to validating the content of individual components via zero or more validators. Standard and customized validators can be matched to UI components in the following way.

The following JSTL code associates a username field with the LengthValidator, which checks that at least six characters are entered into the field.

```
<h:inputText id=”username” value=”#{logonBean.username}”/>
  <f:validateLength minimum=”6”/>
</h:inputText>
```

In the next JSTL code the LongRangeValidator ensures that the value is convertible to the long type and that it is in the given range.

```
<h:input_number id="zip" formatpattern="#####" size="5">
  <validate_longrange minimum="50000" maximum="10000"/>
</input_number>
```

6.2.3. Error Handling

JavaServer Faces has no general application wide support for error and/or exception handling, but such functionality might be added in future releases. Unlike Jakarta Struts, JavaServer Faces is mostly a UI component framework and was not designed to be able to handle a vast range of general J2EE Exceptions.

Similar functionality can be simulated by providing a general <navigation-rule/> element which maps to all view-ids and providing <navigation-case/> elements with appropriate <from-outcome/> elements, which are matched against the action results.

6.2.4. Internationalization

Similar to Jakarta Struts also JavaServer Faces relies on Java Standards for Internationalization issues and especially the Internationalization support already available in the Servlet, JSP and JSTL implementations.

All error messages related to form validation and other components are available as localized application messages and can be overwritten by switching the javax.faces.Messages ResourceBundle.

6.2.5. Security

JavaServer Faces components have no special support for security issues as those are those are generally issues of the application and not the user interface domain. Standard Java Support for Security can be used in a similar way like in Jakarta Struts.

6.3. Evolution

As the JavaServer Faces specification is not yet in its final stage it is difficult to speak of any evolution. Thus this aspect won' be discussed.

7. Apache Cocoon

Apache Cocoon is an XML publishing framework which is built around two strong modern software development concepts: *Separation of Concerns* (SoC) and *Component Oriented Programming* (COP). Instead of providing another multi-tier architecture to separate different application concerns, the separation of concerns is provided through a component pipeline model. Each element of the pipeline specializes in a certain operation and is independent of previous and successive operation stages.

The pipeline elements are implemented as reusable components that are managed by *Apache Avalon*, an Aspect-Oriented (AOP) component framework, also maintained by the Apache Software Foundation. While most of the other Web Tier frameworks rely on Sun's component frameworks such as Servlets or Java Beans, Apache Cocoon uses its own, which makes it difficult to merge it with other Web Tier frameworks.

Cocoon can be used as a Servlet to generate XML / XHMTL content on the fly, or as a command line application to generate static files from various data sources. Declarative application design, support for different output formats, direct support for XSL-T transformations and an included caching model are Cocoon's strong sides. Its weak sides are a component framework that is difficult to learn, a huge code base, no standard application design model and no direct cooperation with Java's component models, Java Beans and Enterprise Java Beans.

7.1. Architecture

The architectural descriptions are based on Cocoon 2.0, as this version can be seen as the foundation for the core concepts implemented in Cocoon. The architectural description is divided into the sections Sitemap, Components and Apache Avalon. The new aspects of Cocoon 2.1, such as advanced control flow, will be addressed in the evolution section, as they are quite independent of the core pipeline model.

7.1.1. Cocoon Sitemap

The sitemap can be seen as the *Front Controller* implementation of Cocoon. It is an XML file that describes declaratively how XML processing pipelines are matched to certain URL patterns. XML data is streamed through the Pipeline via the SAX interface, a streaming API for XML, serialized into text and then sent back to the browser.

Every pipeline begins with a *Generator* which generates an XML stream from an application action, file content or external data sources. The intermediate steps of the pipeline are covered by zero or more *Transformers*, which transform XML content and the final element is a *Serializer*. Beside these elements each pipeline may define its own error handling policies. *Matchers* and *Selectors* are additional components that make it possible to declare matching and selecting policies declaratively in the Sitemap file. Pipelines can be aggregated into hierarchies which are able to cover the complete presentation logic of a web application.

The general structure of a Sitemap file will be presented in the following. The focus will be on the declarative aspects and not on the implementation aspects, as the main components will be described in detail in the following section.

```
<?xml version="1.0"?>
<map:sitemap
        xmlns:map="http://apache.org/cocoon/sitemap/1.0">
```

<map:sitemap/> is the root element. It includes <map:components/>, <map:views/>, <map:resources/>, <map:action-sets/> and <map:pipelines/> as child elements.

```
  <map:components>
```

<map:components/> includes declarations of the components which are used in the Pipelines. The child elements <map:generators/>, <map:transformers/>, <map:serializers/>, <map:readers/>, <map:selectors/>, <map:matchers/> and <map:actions/> are groupings of component declarations.

```
    <map:generators default="file">
      <map:generator name="file"
          src="org.apache.cocoon.generation.FileGenerator"/>
        ...
    </map:generators>
```

As mentioned before every Pipeline begins with a *Generator*, which initializes the SAX stream. The example component uses the content of a file to initialize the stream.

```
    <map:transformers default="xslt">
```

```
    <map:transformer name="xslt"
      src="org.apache.cocoon.transformation.TraxTransformer">
      <use-request-parameters>false</use-request-parameters>
      ...
    </map:transformer>
        ...
  </map:transformers>
```

A *Transformer* transforms an SAX stream from one format into another. The example component is an XSL-T transformer, which uses an XSL-T stylesheet to control the transformation.

```
<map:serializers default="html">
  <map:serializer name="html" mime-type="text/html"
    src="org.apache.cocoon.serialization.HTMLSerializer">
    <doctype-public>-//W3C//DTD HTML 4.0 Transitional//EN
    ...
  </map:serializer>
  ...
</map:serializers>
```

The final component of every pipeline is a *Serializer* which transforms the SAX stream into a binary or char stream.

```
<map:pipelines>
  <map:pipeline>
    <map:match pattern="**book-**.xml">
      <map:generate src="xdocs/{1}book.xml"/>
      <map:transform src="stylesheets/book2menu.xsl">
        <map:parameter name="use-request-parameters" value="true"/>
        <map:parameter name="resource" value="{2}.html"/>
      </map:transform>
      <map:serialize type="xml"/>
    </map:match>
  </map:pipeline>
</map:pipelines>
```

Pipelines are the processing declarations in the Sitemap. Instead of describing all possible elements which can be utilized inside a <map:pipeline/> element, an

example Pipeline is given. The <map:match/> element matches uses the Cocoon variant of wildcard matching where ** escape symbol includes also slash (/) characters in contrast to simple wildcard matching (*).

<map:generate/> uses the content from the file that matches the given expression. {1} matches the first wild card match in the covering <map:match/> element. <map:transform/> is then used to transform the initialized XML stream via the "book2menu.xsl" stylesheet. Finally the content is serialized into XML in char format. [Cocoon UserDocs, 2002]

7.1.2. Cocoon Components

In this section the individual components of Cocoon will be described in detail. The components are Pipelines, Generators, Transformers and Serializers as basic processing blocks, Selector and Matchers as declarative equivalents of conditional logic and finally Actions, Views and Resources as additional components.

A *Pipeline* represents the XML processing chain and is equivalent to the concept of *Servlet chaining*. Every Pipeline begins with a Generator, has zero or more intermediate processing steps represented by Transformers and ends with a Serializer. In addition to the main processing components, Views can be inserted to intercept the processing at certain steps, Selectors and Matchers to integrate conditional logic into the Pipeline and Aggregator elements to combine content from different sources.

The streams can be cached persistently or into memory. In the latter case, a lightweight variant of the DOM interface for XML is used to serialize SAX streams. Cocoon' scaching mechanisms can be used as a variant of *Session handling* to store various XML informations locally in memory, but the main purpose is speed enhancement, as XSL-T transformations can be quite slow if not optimized. As Cocoon is especially used for cases where heterogeneous XML content needs to be aggregated it is not uncommon to utilize more than ten XSL-T transformations to process a single request.

The *Generators* represent the initial points of the pipelines. They are responsible for initiating SAX streams and delivering them to the next stage in the pipeline. Although the FileGenerator is the most common and the default, it is not the only one. Other generators are the DirectoryGenerator, which streams a directory structure in XML form, the FragmentExtractorGenerator, which streams an XML subtree, the Velocity-, JSP- and PHPGenerator as connection points to scripting languages and various others.

The intermediate stages of a pipeline are then covered by *Transformers*. The default transformer is the XSLTTransformer which transforms the

incoming SAX stream according to the templates declared in an XSL-T stylesheet. Other transformers are the FragmentExtractorTransformer, the SQLTransformer to create SAX streams from SQL query results, the FilterTransformer, to perform simple filtering on the stream and Xinclude- / CIncludeTransformers to resolve XML include statements and replace them with the linked content.

Transformers which emulate standard Java behaviour are the ReadDOMSessionTransformer and WriteDOMSessionTransformer for Session handling, the i18nTransformer for internationalization and the LogTransformer for logging. These components do not use the Java (J2SE / J2EE) standards for Servlet Session handling, internationalization or logging which makes it difficult to run Cocoon parallel to other User Interface technologies (JavaServer Pages, Java Swing, ...) where logging and/or internationalization declarations should be shared.

The final processing stages are covered by *Serializers*. The HTMLSerializer is the default component for serialization, but support for other output formats is also available. Examples of other Serializers are the XHTMLSerializer, the XMLSerializer, the TextSerializer to cover other ASCII/XML formats, the PDFSerializer, PSSerializer and PCLSerializer to cover binary text formats and the SVGSerializer and VRMLSerializer cover these two image formats.

Beside the basic processing blocks *Selectors* and *Matchers* can be used to include conditional logic into the processing. These elements are most commonly used to match processing blocks against some combination of environment variables (URI, headers, cookies etc.). While Matchers only allow for simple "yes/no" decisions, Selectors can be used to describe rather complex conditional cases. The equivalent of these elements in procedural programming languages are "if-else" and "switch-case" constructs. The syntax of the Selector declarations has been influenced by the <xsl:test/> statement of XSL-T stylesheet language.

Because a Pipeline architecture with static URL matching is not really suitable for more complex web applications, *Actions* can be used as initial processing steps whose outcome determines which pipeline will be used to process the request or if the currently pipeline will continue with request processing.

This approach doesn't seem too elegant, as it breaks the linear pipeline processing, but it is a highly flexible tool to switch pipeline mappings based on runtime parameters. The equivalent of Actions in Jakarta Struts is the *Service-To-Worker* pattern, where URL are mapped to database and other actions whose outcome determines which view to select.

An alternative to actions are *XSP pages* (eXtended Server Pages). They are mainly a variant of JavaServer Pages with stricter XML syntax and logicsheets to mimic the concept of taglibs. XSP pages can be used to assemble basic abstract XML pages with included logic to server as pipeline starting points. The main difference between JSP and XSP is that while XSP are alternatives to actions and generators to initiate request processing chains, JSP are commonly used as the final elements in processing chains, at least in their normal deployment contexts.

To debug XML transformations or more generally to provide consistent views of intermediate transformation stages Cocoon provides *View* components which are declared for the whole Sitemap. Views can be used to intercept the current processing and make it available through a certain URL pattern.

Finally *Resources* act as shortcuts to Pipelines that can be used from within other Pipelines to redirect to error messages or other application wide messages and pages. The equivalents in traditional web programming are internal redirects. Like redirects resources act as exit points to Pipelines. [Cocoon UserDocs, 2002]

7.1.3. Avalon Component Framework

All Cocoon components are based on the Avalon framework, and *Aspect-Oriented* (AOP) component framework of the Apache Software Foundation. Apache Avalon provides a platform including various component containers, utilities, tools and default components.

Component Oriented Programming (COP) is the main paradigm of the Avalon framework. Component Oriented Programming can be seen as an enhancement of the Object Oriented Programming (OOP) programming paradigm. Components have stricter interfaces than classes and rely on assembling and composition for application development instead of inheritance. The concept of *Black-box reuse* is often quite strong in component technologies.

Normally containers provide lifecycle and deployment management for components. Components are assembled to provide application blocks which can be used for variety of deployment purposes. The Java 2 Enterprise Edition has Enterprise Java Beans (EJB) as its standard component model, which is most suitable for complex multi-tier architectures with high demands of security, interoperability and transaction safety.

Compared to other Java component technologies Avalon components have a much higher grade of reuse. Instead of using fixed interfaces for inter-component communication Avalon components rely on the concept of *Inversion*

of Control (IoC) for communication. As mentioned before Inversion of Control relies on the Observer pattern for communication which reduces static coupling and fixed interfaces.

Separation of Concerns (SoC) is another technique which is used in the Avalon framework. Components are classified by their roles instead of their internal functionality. The base classes and interfaces of the Avalon framework will be presented later. Separation of Concerns is quite similar to the concept of Aspect Oriented Programming (AOP), but the implementation technique in Avalon is different from aspect-oriented languages. Avalon relies on standard Java language idioms such as interfaces and listeners to achieve similar results as aspect-oriented languages like AspectJ.

The Avalon components used by Cocoon are ComponentManager, Composable, Component, Configuration, Configurable and ConfigurationBuilder. The main focus is on *Dynamic Composition* and *Runtime Configuration*, which are essential for such a complex application like Apache Cocoon. ComponentManager, Composable and Component focus on mechanisms such as dispatching, composition and procedural functionality whereas Configuration, Configurable and ConfigurationBuilder are related to the aspect of configuration. [Cocoon DevelDocs, 2002]

7.2. Concerns

In the case of Apache Cocoon the aspect of Security is omitted from the set of concerns, as it is not explicitly dealt with in Cocoon.

7.2.1. Navigation

Apache Cocoon doesn't provide any direct support for Navigation, although it is possible to define navigation models via XML and use them to create HTML mark-up for breadcrumb, menu and tab navigation like for example in the Navigation framework of the Apache Lenya project [ApacheLenya Navigation, 2003].

7.2.2. Form Validation

Apache Cocoon offers two different ways to deal with form validation. *XMLForm*, the first way, is oriented on the W3C XForms working draft. XForms is a standard for platform and display independent representation of input forms. *FormValidatorAction* is the simpler and older way and is promoted as the standard way to deal with form validation. The FormValidatorAction is the Cocoon equivalent of the ActionForm in Jakarta Struts.

To validate user input the FormValidatorAction is placed in the pipeline. The constraints of the form validation are specified in a descriptor file. The results of the form validation can either be used in an XSP through the formval logicsheet or be displayed with the SimpleFormTransformer, which is the preferred way, as it is fits conceptually better into the pipeline model.

The following example illustrates the usage of the FormValidatorAction in combination with XSP server pages. For each validation outcome, acceptance and refusal, one XSP page is used.

```
<map:match pattern="car-reservation">

  <map:act type="form-validator">
    <map:parameter name="descriptor" value="descriptor.xml"/>
    <map:parameter name="validate-set" value="car-reservation"/>

<!-- acceptance of form input -->

    <map:generate type="serverpages" src="OK.xsp"/>
    <map:transform src="stylesheets/dynamic-page2html.xsl"/>
    <map:serialize/>
  </map:act>

<!-- refusal of form input -->

  <map:generate type="serverpages" src="test/ERROR.xsp"/>
  <map:transform src="stylesheets/dynamic-page2html.xsl"/>
  <map:serialize/>

</map:match>
```

The next example utilizes the SimpleFormTransformer to validate the input form.

```
<map:match pattern="car-reservation">

  <map:act type="req-params">
    <map:parameter name="parameters" value="order"/>

    <map:act type="form-validator">
```

```
      <map:parameter name="descriptor" value="descriptor.xml"/>
      <map:parameter name="validate-set" value="car-reservation"/>

<!-- acceptance of form input -->

      <map:generate type="file" src="OK.xml"/>
      <map:transform src="stylesheets/dynamic-page2html.xsl"/>
      <map:serialize/>
    </map:act>

  </map:act>

<!-- refusal of form input -->

  <map:generate type="file" src="test/ERROR.xml"/>
  <map:transform src="stylesheets/dynamic-page2html.xsl"/>
  <map:transform type="simple-form"/>
  <map:serialize/>

</map:match>
```

The second example is slightly longer, but has the advantage that it stays completely in the XML domain and doesn' tneed to have any validation logic be described in Java or XSP form.

7.2.3. Error Handling

Error Handling in Apache Cocoon is integrated into the pipeline model. Java Exceptions can be handled like any other data in the pipeline. <map:handle-errors/> elements are used to define elements that are used to serialize a pipeline stream in case any Java exceptions occur. If <map:handle-erros/> declarations are omitted Exceptions are forwarded to the Servlet level and cause ServletExceptions.

Error handling declarations can be made for the Sitemap or Pipeline scope. <map:handle-errors/> elements can include any Sitemap components except for a Generator, because a standard ErrorGenerator is used to stream data from the Java exception.

The following example shows an excerpt of the default Cocoon Sitemap where a user-friendly error-page is created through a Transformer and Serializer.

```
<map:handle-errors>
  <map:transform src="stylesheets/system/error2html.xsl"/>
  <map:serialize status-code="500"/>
</map:handle-errors>
```

7.2.4. Internationalization

The aspects of Internationalization (i18n) and Localization (l10n) are addressed in a non-standard way in Apache Cocoon to fit into the XML pipeline model. Apache Cocoon uses the I18nTransformer in combination with multilanguage XML dictionaries to convert abstract message keys into localized messages.

The features of the i18n Transformer are text translation attribute translation, parameter substitution and date, number and currency formatting. As the restricted declarative approach to internationalization and localization doesn' tfit more demanding needs, XSP is suggested as an alternative where more complex formatting is needed.

7.3. Evolution

As the release history of Apache Cocoon is not well documented, I will choose arbitrary release versions to exemplify the evolution of the Apache Cocoon code base and project.

The presented release version are 1.0, 1.5., 2.0 and 2.1. The aspect of Advanced Control Flow in Cocoon 2.1 will be discussed in detail as it has not been presented in previous sections.

7.3.1. Cocoon 1.0

Cocoon 1.0 is based on a simple Servlet with FactoryMethod implementations for XML Parsers and Processors. The processing of XML is based on DOM. The project source code is represented by 14 rather simple main classes. This release is better described as a prototype or demo than a mature working system.

7.3.2. Cocoon 1.5

In Cocoon 1.5 a clear architecture begins to emerge. The FactoryMethod implementations are being replaced by a component framework, which will be externalized into the Apache Avalon framework at a later stage. The

component palette is now divided into XML Parsers, Producers, Processors and Formatters. XML Processing is still done over the DOM interface. The source code consists of about 80 main classes.

7.3.3. Cocoon 2.0

In the next major release, Cocoon 2.0 (2.0.4), the component framework has been externalized into the Avalon framework and SAX is used to stream the XML information through the pipeline. This release utilizes code from 34 other Java projects.

The number of components used for this release is huge, as Cocoon tries to hide most of the aspects from the underlying Servlet framework. The most important additions to the component palette are Generators, Actions, Selectors and Matchers.

7.3.4. Cocoon 2.1

The most important new feature in Cocoon 2.1 is Advanced Control Flow. This feature can be best described in contrast to an ordinary interaction model for a web application which can be seen as a finite state machine (FSM), an application which is described by a set of possible actions, states and state-transitions.

While Jakarta Struts and JavaServer Faces use a variant of a finite state machine, older versions of Cocoon don't represent any concrete interaction model, but something that resembles a vague Model 2 variant instead. The feature of advanced control flow introduces an interaction model into Cocoon.

The control flow is described using a high level language such as JavaScript in an ordinary function, which allows developers to design the interaction model like an ordinary program. State persistence between HTTP requests is provided through a framework which stores the exact execution environment of a certain function and allows the continuation of the function with the same parameters.

I omitted a more detailed discussion of advanced control flow from the presentation of Apache Cocoon as it presents a very different web application model from the traditional pipeline model. Advanced control flow promotes a more flat application design, where all the business logic is described in one single function where more specific presentation tasks are pushed into pipeline streams.

8. Java Portlet Specification

The Portlet Specification is based on a Java Community Process (JCP) to standardize different approaches on defining a Portal framework on top of the Servlet framework. A Portlet is a web component which is conceptually between a Servlet and a JavaServer Faces component. Portlets are aggregated in Portal services as pluggable user interface components with included connections to the EIS or RDBMS backend. Figure 8.1 illustrates the relationships between the client device, the Portal server and the Portlet container in the creation of a portal page.

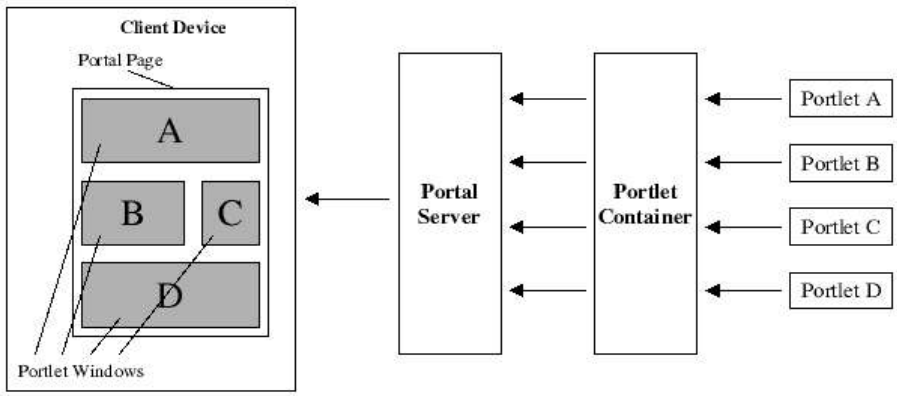

Figure 8.1: Portal page creation in the Portlet framework [Sun JavaPortlet, 2003]

Portlets generate content fragments which are usually markup (e.g. HTML, XHTML, XML) and are then used in the Portal to compose a complete page. A Portlet Container is responsible for the lifecycle management of a portal. The user interacts with the portal which then forwards the requests to individual Portlets.

8.1. Architecture

The architectural description of the Java Portlet Specification focuses mainly on the additions the Portlet framework has to offer to the Servlet framework such as initial support for User information. The description is divided into a general comparison of the two frameworks, an overview of Portlet deployment and configuration issues and a discussion on the support for user information in the Portlet framework.

8.1.1. Comparison between Servlets and Portlets

The Portlet Specification is based on the Servlet Specification v 2.3 and many architectural issues are directly related to it. The main similarities are that both

are Java technology based web components, are managed by a specialized container, generate dynamic content and use a request / response paradigm to interact with a web client.

The main differences are that Portlets generate only fragments and not complete responses, they can't be accessed directly through a URL, web clients interact with Portlets through a Portal system, the request handling of Portlets is different from the HTTP requests used by Servlets, the Portlets include information about their access mode and window states.

The extra functionality of Portlets is that they have means for storing persistent configuration and customization data, have access to user profile information and have URL rewriting functions. The main restrictions are that Portlets can neither set character set encoding nor the HTTP headers of the response and don' thave any access to the URL of the client request to the Portal.

8.1.2. Deployment and configuration

Portlets are deployed and configured in a similar way like Servlets. A PortletConfig object provides initialization information which is fed into it through the Portlets XML Deployment Descriptor.

Information about UI messages can be declared either directly in the Deployment Descriptor or indirectly via a resource bundle. Those messages are the purpose, title, short-title and keywords. The resource bundle approach is preferable, as it allows the use of localized messages. The following example shows how this information can be declared inline.

```
<portlet>
  ...
  <portlet-info> 20
    <title>Stock Quote Portlet</title>
    <short-title>Stock</short-title>
    <keywords>finance,stock market</keywords>
  </portlet-info>
  ...
</portlet>
```

In the case that the messages are declared in a resource bundle, the name of the resource bundle has to be declared in the deployment descriptor. The next example shows a deployment descriptor excerpt with a resource bundle reference.

```
<portlet>
  ...
  <resource-bundle>
    com.foo.myApp.QuotePortlet
  </resource-bundle>
  ...
</portlet>
```

8.1.3. User Information

As customization and personalization are the main aspects that differentiate Portlets from Servlets, these are also addressed in the specification. The responsibility to expose user information to the Portlets is on the side of the Portlet Container.

Unfortunately the JCP which was concerned with the Portlet Specification had not standard Java API for user information. The declaration of user information in the deployment descriptor was chosen as an initial variant for testing purposes. The following excerpt shows the declaration of user information in the deployment descriptor.

```
<portlet-app>
  ?
  <user-attribute>
    <description>User Given Name</description>
    <name>user.name.given</name>
  </user-attribute>

  <user-attribute>
    <description>User Last Name</description>
    <name>user.name.family</name>
  </user-attribute>

  ...
</portlet-app>
```

In my opinion the User Interface part of the Portlets could have been handled well via JavaServer Faces components and the Java Community Process related to the Portlet Specification should have concentrated on the issues which are

important for Portals such as customization of menus, visual composition and user-specific styles instead of defining a new component architecture which doesn't offer much new to the Servlet / JSP / JSF combination.

8.2. Concerns

In the list of concerns of the Portlet framework Navigation and Form validation are omitted as they are concerns of lower level mark-up rendering elements such as JSP pages or JavaServer Faces components.

8.2.1. Error Handling

Error Handling in the Portlet framework is mostly the responsibility of the Portlet container. Portlets can throw three types of Exceptions: *PortletExceptions*, *PortletSecrurityExceptions* and *UnavailableExceptions*.

A *PortletException* is a general exception which signalizes that an error has occurred in the processing of the request. In case such an exception is thrown by a Portlet the Portlet container should clean up the request and continue processing the other Portlets. In case a *PortletSecurityException* is thrown, the specification doesn't specify any concrete actions the Portlet container should carry out. This kind of exceptions is thrown in case the user does not have sufficient rights to access the requested resource.

An *UnavailableException* signalizes like its name suggests, that the Portlet is temporarily or permanently unavailable. I a permanent unavailability is indicated, the Portlet container has to remove the Portlet immediately. Such a Portlet is to be considered unavailable until the Portlet container restarts. *RuntimeExceptions* thrown from external object such JavaServer Pages, JSF components or Servlets have to be wrapped into PortletExceptions.

8.2.2. Internationalization

Internationalization and localization in the Portlet framework are based on Java standards such as ResourceBundle and Message keys. As Portlets act as small-scale service containers the act of creating displayable mark-up elements is mostly delegated to JavaServer Pages and JSF components.

8.2.3. Security

Security Handling in the Portlet framework is mostly based on the security facilities of the Servlet framework. The Servlet framework provides basic authentication and role-based authorization mechanisms that can be used from within Servlet and Portlets.

It is the responsibility of the Portlet container to inform the Portlet of the roles the user is in. Authentication is not handled by the Portlet container; it is left completely to the underlying Servlet container.

Security role mappings are declared in the deployment descriptor. While this suggests, that the Portlet framework supports some kind of declarative security like in the Servlet framework, it is just a declarative mapping of security roles to references used in the Portlet code, and is therefore programmatic.

The following example shows how the security role reference "FOO" is mapped to the role-name "manager".

```
<portlet-app>
  ...
  <portlet>
    ...
    <security-role-ref>
      <role-name>FOO</role-name>
      <role-link>manager</manager>
    </security-role-ref>
  </portlet>
  ...
  ...
</portlet-app>
```

8.3. Evolution

As the Portlet specification is also quite young with the final version released in October 2003 it is also in this case difficult to speak of an evolution at this point.

9. Discussion

When writing this thesis, I realized that while it certainly mattered what kind of primary architecture and paradigm was chosen for a framework, the business environment mattered sometimes even more. While the two presented frameworks of the Apache Software Foundation provide two very pragmatic and clear models, the frameworks of Sun Microsystems are much more bound to business goals and less bound to clear paradigms or models.

In the following some of the project related issues of the frameworks will be discussed. Addressed issues are code reuse in other projects, future scenarios, threats from other projects etc. Deployment issues and possible application scenarios of the frameworks won't be discussed.

9.1. Jakarta Struts

While Jakarta Struts has been one of the most popular web application frameworks for business applications, the background is nevertheless non-commercial. As the object-oriented paradigm and its application in Java application design has changed considerably since the introduction of Java in 1995 so has the use of design patterns in the J2EE platform. The Jakarta Struts framework with its BluePrints Front Controller implementation is not excluded from this change.

Most of the presentation parts of Jakarta Struts are continuously replaced by official J2EE technologies such as the JSTL and JSF for the Struts taglibs. One possible future scenario of the Jakarta Struts project is to concentrate on the architecture part by providing a working infrastructure for the various J2EE web technologies such as Portlets, Web Services and JavaServer Faces components.

Concrete threats to the survival of the Jakarta Struts framework are JavaServer Faces for the presentation part and the Spring framework for the integration of different J2EE components. As big parts of the Struts code base have been ported into the Jakarta Commons project which is used in a large number of Open Source projects, a high amount of Jakarta Struts code is recycled in other projects.

9.2. JavaServer Faces specification

The JavaServer Faces framework is a good example of how business constraints determine the final architecture. While the architecture is inspired by previous Model 2 architectures like Jakarta Struts the implementation with the focus on user interfaces aims to compete with the .Net WebForms framework and tries

to open the field of web application design for Java Swing developers by bringing the desktop interaction model into the web domain. Exploration of new market areas for WYSIWYG (What-You-See-Is-What-You-Get) web application design is one of the main goals.

If Sun with its Java Studio Creator IDE and IBM with its Eclipse IDE manage to provide high quality plugins to design JavaServer Faces components and JSP pages in an similar way like in the Microsoft Visual Studio .NET, JavaServer Faces is a sure candidate for a J2EE web application standard.

As at least two OpenSource implementations of the JavaServer Faces frameworks are already available, there is also the possibility that Sun continues to maintain the JavaServer Faces specification while the reference implementation is provided by an external project like in the case of many Apache projects.

9.3. Apache Cocoon

A niche for declarative J2EE application design with a focus on XML processing is covered by the Apache Cocoon project. While the pipeline model is an easy to grasp and elegant approach to build publication centred web applications, the advanced control flow feature of the 2.1 release is a rather exotic and experimental feature which contrasts strongly with otherwise declarative application design in Cocoon and the Model 2 variants used in Jakarta Struts and JavaServer Faces.

While Apache Cocoon provides a very fresh approach to web application design it leaves too many J2EE standards unused to become a real J2EE mainstream technology. Its weak support for JavaServer Pages and JavaServer Faces technologies will draw developers who want the flexibility of XML and the ease of JSF probably to some other technological solutions such as the Model 2X approach of Orbeon [Orbeon, 2004]. As the JSP taglibs are providing already many tools to manage XML DOM tress easily from a JSP page, the cases where a pure XML technology such as Apache Cocoon would prove to be useful, are becoming quite rare.

Another J2EE standard Apache Cocoon could be more interoperable with is the Portlet framework. Cocoon could act as an architectural basis for declarative XML based Portlet design, as portals are in general an application domain where Cocoon is quite strong.

While the purpose OpenSource frameworks in J2EE is of course to provide alternatives to the mainstream Java technologies promoted by Sun, most of the successful J2EE projects fill clear gaps in the J2EE technology stacks and reuse as much infrastructure as possible. While Cocoon is quite unique in

its declarative approach, it reproduces too much infrastructure which is already present in the base Java architecture such as logging, internationalization, XML persistence, Servlet framework etc.

9.4. Java Portlet specification

The Java Portlet specification aims to define a common standard for Portlet development / deployment on the J2EE platform. The strengths of the specification are the strong adherence to already available J2EE standards such as the Servlet framework for the Portlet container infrastructure and JSP as well as JSF for markup generation.

The weaknesses are mainly that the Portlet framework doesn't offer much more than the Servlet framework and adds another container technology to the J2EE platform. Enterprise Java Beans are commonly used to model business state and JSF components can be used to model User Interface state. Portlets as service components between the core services defined in EJBs or other Business Tier technologies and fine grained JSF components can be useful as a component technology for Portal applications, but adds in my opinion an unnecessary amount of complexity to the already difficult to grasp API set of the J2EE platform.

As the JCP Portlet specification is already being adopted as a J2EE standard for Portlet development, the survival as a J2EE standard seems to be guaranteed. A Java standard for the specification of user information would make the use of the Portlet framework easier.

9.5. Alternatives to J2EE Web tier frameworks

While the presented Web tier frameworks represent the main technique to enhance the functionality of the Servlet framework and to ease its use, there are alternatives. On the framework side *multi-tier frameworks* such as Spring [SpringFramework, 2004] are currently en vogue. They focus more on integration of different J2EE technologies into web applications than concentrating on the special needs of one tier only.

With the advance of .NET technologies and an ongoing effort to port successful OpenSource Java projects to the .NET platform there is also the possibility that some Web tier frameworks will be ported. The use of multi-platform frameworks would ease the development for multiple platforms by hiding the platform dependent issues.

While frameworks can be seen as the most popular extensions mechanism to platform infrastructure, the use of scripting languages can be sometimes more efficient. Examples of object-oriented scripting languages which have

been ported to the JVM are JavaScript, Python and Ruby. While JavaScript does not have a standard API set and is mostly used for scripting web pages, Python and Ruby are extremely popular dynamically typed languages which are used for various demanding tasks such as system administration, prototyping, scientific computing as well as web application design.

10. Summary

This thesis describes four different J2EE Web tier frameworks with a focus on their high-level architecture and addressed concerns. The described frameworks are Jakarta Struts and Cocoon from the Apache Software Foundation and JavaServer Faces as well as the Portlet framework from Sun Microsystems.

The introductory parts give a basic introduction to frameworks in the first chapter and an introduction to the J2EE Web tier in the second chapter. The presented issues are the general application model, the Servlet framework, the J2EE BluePrints and the purpose of the J2EE Web tier frameworks.

In the next four chapters the frameworks are described. Each of these chapters is divided into three parts: the architecture, the concerns and the evolution of the framework. For JavaServer Faces and the Portlet framework the evolution parts are left out, as both frameworks are too young to be discussed from an evolutionary point of view. The presentations don't aim to prepare any evaluation as the presented frameworks are architecturally very different and mostly address different issues.

The next chapter acts as a discussion of the results and focuses mainly on issues related to project and business concerns to address also non-technical issues of the frameworks briefly. It also expresses some evaluation on the usability of the frameworks and draws possible future scenarios for the specific frameworks. Finally alternative ways to extend the J2EE Web tier are presented.

All in all the thesis tries to present a high level overview on the architecture and evolution of the J2EE Web tier by presenting its architectural base, which consists of the Servlet framework and J2EE BluePrints, and the framework extensions, their architecture, interrelations and addressed application concerns.

References

[ApacheAvalon, 2004] *Apache Avalon – Overview*, 1997-2004, The Apache Avalon Project
http://avalon.apache.org/index.html

[ApacheLenya Navigation, 2003] *Apache Lenya – Concepts and Best Practices – The Navigation Framework*
http://www.wyona.org/docs/concepts/navigation.html

[Bruchez BothWorlds, 2003] E. Bruchez, O. Tazi, *Integrating JSP/JSF and XML/XSLT: The Best of Both Worlds*, 2004 TheServerSide.com
http://www.theserverside.com/articles/article.jsp?l=BestBothWorlds

[Cann, 2003] S. Cann et al., *Navigating the Application Development Frameworks Terrain*, JavaOne 2003
http://www.mvc2frameworks.org

[Cocoon UserDocs, 2002] *Apache Cocoon 2.0 – User Documentation*, 1999-2002 The Apache Software Foundation.
http://cocoon.apache.org/2.0/userdocs/index.html

[Cocoon DevelDocs, 2002] *Apache Cocoon 2.0 – Developer Documentation*, 1999-2002 The Apache Software Foundation.
http://cocoon.apache.org/2.0/developing/index.html

[Cunningham & Cunningham, 2004] *The offical Wiki Version of the history of the JavaLanguage*. WikiWikiWeb. Cunningham & Cunningham, Inc.
http://c2.com/cgi/wiki?JavaHistory

[Emmerich, 2002] W. Emmerich, *Distributed Component Technologies and their Software Engineering Implications*, ICSE ' 02 1925 May 2002 , 2002 ACM

[Fayad, 1997] M. Fayad, D. C. Schmidt, *Object-Oriented Application Frameworks*, Communications of the ACM, Vol. 40, No. 10, October 1997.

[Gamma, 1994] E. Gamma et al., *Design Patterns – Elements of Reusable Object-Oriented Software*, 1995 Addison-Wesley Publ.

[Hunter, 1998] J. Hunter, W. Crawford. *Java Servlet Programming*. 1998 O' Reilly & Associates, Inc.

[Hunter, 2002] J. Hunter. *Servlet Best Practices, Part 1* (from Java Enterprise Best Practices). 2002 O'Reilly & Associates, Inc. http://www.onjava.com/pub/a/onjava/excerpt/jebp_3/index1.html

[JakartaStruts ,2003] *The Struts Users' Guide*, 2000 – 2003 Apache Software Foundation http://jakarta.apache.org/struts/userGuide/index.html

[Koskimies, 2003] K. Koskimies. *Ohjelmistoarkkitehtuurit (Software Architectures) – Lectures*. 2003 Tampere University of Technoology

[Mattsson, 1996] M. Mattsson. *Object-Oriented Frameworks – A survey of methodological issues*, 1996 University College of Karlskrona/Ronneby, Sweden

[Mazzocchi, 2003] S. Mazzocchi. *Apache Cocoon – History*. 2003 Apache Software Foundation http://cocoon.apache.org/history.html

[O'Reilly, 2002] *Java API Map*, 2002 O' Reilly & Associates, Inc. http://www.onjava.com/pub/a/onjava/api_map

[Orbeon, 2004] *Orbeon : Model 2X*, 1999-2004 Orbeon, Inc. http://www.orbeon.com/model2x/

[Rossi, 2001] G. Rossi et al., *Designing Personalized Web Applications*, WWW10 1-5 May 2001

[SpringFramework, 2004] Java/J2EE Application Framework http://www.springframework.org/index.html

[Sun J2EEPatterns, 2002] *Core J2EE Patterns*, 2001-2002 Sun Microsystems, Inc. http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html

[Sun J2EEModel, 2004] *J2EE Overview - Application Model*, 1994-2004 Sun Microsystems, Inc.

http://java.sun.com/j2ee/appmodel.html

[Sun JavaHistory, 2000] *The Java Platform: Five Years in Review*, 2000 Sun Microsystems, Inc.
http://java.sun.com/features/2000/06/time-line.html

[Sun JavaOverview, 2004] *Java Technology Overview*, 1994-2004 Sun Microsystems, Inc.
http://java.sun.com/overview.html

[Sun JavaPortlet, 2003] A. Abdelnur, S. Hepper. *Portlet Specification – Version 1.0 – Public Review Draft*. 2003 Sun Microsystems, Inc.

[Sun JavaServerFaces, 2003] C. McClanahan, Ed Burns. *JavaServer Faces - Specification 1.0 – Proposed Final Draft*. 2003 Sun Microsystems, Inc.

[Sun JavaServerPages, 1998] *JavaServer Pages - Specification 0.92*. 1998 Sun Microsystems, Inc.
http://www.kirkdorffer.com/jspspecs/jsp092.html

[Sun JavaServer Pages, 2004] *JavaServer Pages[tm] Technology - White Paper*, 1994-2004 Sun Microsystems, Inc.
http://java.sun.com/products/jsp/whitepaper.html

[Sun JSTL, 2003] Delisle, Pierre (ed.), *JavaServer Pages Standard Tag Library – Version 1.1*, November 2003 Sun Microsystems, Inc.

[Sun Servlet, 2001] *Java Servlet 2.3 and JavaServer Pages 1.2 Specifications – Final Release*, September 2001 Sun Microsystems, Inc.

[Szyperski, 1997] C. Szyperski, *Component Software – Beyond Object Oriented Programming*, Addison-Wesley 1997

[Takawagi, 2001] O. Takawagi, A. Spender, A. Stevens and J. Bouyssou, *Programming J2EE APIs with WebSphere Advanced.* IBM RedBooks: SG24-6124-00, August 2001

[vanGurp, 2001] J. van Gurp, J. Bosch, *Design, implementation and evolution of object oriented frameworks: concepts and guidelines*, Software – Practice and Experience 2001, Jon Wiley & Sons, Ltd.

[Wafer, 2004] A. Eden, K. Thompson, *Wafer – Web Application Framework Research*
http://www.waferproject.org/index.html

[Yu, 2002] J. Yu, *Struts 1.1 : Should I upgrade?* Aug. 2002, TheServerSide.com
http://www.theserverside.com/articles/article.jsp?l=Struts1_1